IFT 2015 : Devoir 1

Professeure : Esma Aïmeur
Démonstrateurs : Dorsaf Sallami et Hugo Rocha
Hiver 2024

1 Consignes générales

- Date de rendu : mardi 6 février à minuit.
- Le devoir sera noté sur 20 points.
- Ce travail est à faire seul(e) ou à deux.
- Aucun plagiat ne sera accepté.
- Il est possible que des précisions ou des modifications de l'énoncé soient envoyées par courriel et mises sur Studium.
- Chaque jour de retard pour la remise de votre travail entraînera une pénalité de 5 points.
- Tout code est à faire avec Java JDK 17 .
- Vous devez soumettre les fichiers Java demandés et un unique PDF avec les réponses écrites.
- Le PDF peut être des photos/scans de réponses écrites à la main. Attention à la lisibilité !
- Si vous êtes en équipe : l'un soumet le devoir complet et l'autre soumet uniquement un PDF avec le nom de son partenaire.
- L'évaluation du code peut être automatisée, assurez-vous d'avoir les mêmes signatures de fonctions.
- Aucun dépannage ne sera effectué dans votre code. Si votre code ne fonctionne pas, vous n'obtiendrez aucun point.
- Vous avez le droit de créer autant de fonctions intermédiaires que vous voulez.
- Pour toute question sur le TP1, nous vous demandons de la publier sur le forum "TP1 Q&A" sur Studium.
- Bon travail !

2 Analyse de complexité (Total 4 points)

Exercices : Compte tenu des fonctions mystérieuses suivantes, pour chacun d'eux, déterminer quelle est la complexité dans le temps et l'espace de son exécution (Big O) et expliquer ce que vous pensez que la fonction fait. Les réponses simples ne seront pas acceptées, il est nécessaire de justifier votre réponse. Par exemple : Si dans l'exercice la récursion est utilisée, vous pouvez soutenir votre justification en présentant l'arbre de récursion. Toutes vos réponses doivent être incluses dans votre rapport.

1)

```java
public class MistFonction1 {

    3 usages
    public static int mistFonction1(int m, int n) {
        if (m == 1 && n == 1) return 1;
        if (m == 0 || n == 0) return 0;
        return mistFonction1( m: m - 1, n) + mistFonction1(m,  n: n - 1);
    }
}
```

2)

```java
public class MistFonction2 {

    1 usage
    public static List<List<String>> mistFonction2(String target, List<String> pieces) {
        List<List<String>>[] table = new ArrayList[target.length() + 1];
        for (int i = 0; i <= target.length(); i++) {
            table[i] = new ArrayList<>();
        }
        table[0].add(new ArrayList<>());

        for (int i = 0; i <= target.length(); i++) {
            for (String piece : pieces) {
                if (i + piece.length() <= target.length() &&
                        target.startsWith(piece, i)) {
                    List<List<String>> newCombinations = new ArrayList<>();
                    for (List<String> subarray : table[i]) {
                        List<String> newSubarray = new ArrayList<>(subarray);
                        newSubarray.add(piece);
                        newCombinations.add(newSubarray);
                    }
                    table[i + piece.length()].addAll(newCombinations);
                }
            }
        }

        return table[target.length()];
    }
}
```

3)

```java
1 usage
public class MistFonction3 {
    1 usage
    public static boolean mistFunction3(int target, int[] options) {
        boolean[] table = new boolean[target + 1];
        table[0] = true;

        for (int i = 0; i <= target; i++) {
            if (table[i]) {
                for (int option : options) {
                    if (i + option <= target) {
                        table[i + option] = true;
                    }
                }
            }
        }
        return table[target];
    }
}
```

4)

Dans les exercices 2 et 3, on a utilisé un tableau pour sauvegarder les réponses précédentes (plus petites instances du problème) et éviter de faire des calculs inutiles. Cette technique de résolution de problèmes est appelée programmation dynamique. Quelle serait la complexité temporelle de ces implémentations, en utilisant uniquement la récursivité à la place ? Justifiez votre réponse.

3 Piles (Total 16 points)

- L'idée derrière les exercices suivants est d'implémenter 3 piles : une pile simple, une pile double et une pile spéciale, pour comprendre comment cette structure de données se comporte et son utilité pour un certain type de scénario.
- Il faut implémenter chaque pile avec son nom de classe, la pile simple avec le nom « ArrayStack », la pile double avec le nom « DoubleArrayStack » et la pile spéciale avec le nom de classe « SpecialArrayStack ». Leurs interfaces doivent s'appeler « Stack », « DoubleStack » et « SpecialStack ». Vous n'êtes pas autorisé à importer le paquet java.util pour les implémentations « ArrayStack », « DoubleStack » et « SpecialStack ».
- N'oubliez pas de tenir compte de la gestion des erreurs lors de la mise en œuvre des méthodes.
- N'oubliez pas d'ajouter des docstrings et des commentaires à votre implémentation. Suivez également les principes de CamelCase. Le code non documenté sera pénalisé.

3.1 Pile simple (3 points)

Écrire une classe « ArrayStack » qui fonctionne à l'aide d'un tableau générique (n'oubliez pas que vous devez également implémenter son interface appelée « Stack »). Le nombre maximal d'éléments est de 100. Ce sont les méthodes que la pile simple devrait pouvoir exécuter :

Fonction	Signature	Détails
Push	public void push(E e)	Ajoute un élément sur la pile.
Pop	public E pop()	Retire le dernier élément sur la pile et le renvoie.
Top	public E top()	Renvoie le dernier élément sur la pile.
Size	public int size()	Renvoie la longueur de la pile.
Is Empty	public boolean isEmpty()	Vérifie si la pile est vide.
To String	public String toString()	Produit une représentation en chaîne (String) des éléments de la pile classés de haut en bas.

Soumettre les fichiers « Stack.java » et « ArrayStack.java » dans un dossier appelé « Stack » dans Studium.

3.2 Pile double (4 points)

Écrire une classe « ArrayDoubleStack » qui implémente 2 piles dans un même tableau générique (n'oubliez pas que vous devez également implémenter son interface appelée « DoubleStack »). Le nombre maximal d'éléments (longueur pile 1 + longueur pile 2) est de 100.

Cette classe doit avoir toutes les fonctions ci-dessus pour chaque pile, avec pour seule différence un booléen « one » qui indique si l'on traite les éléments à la 1re ou 2e pile.

Faire attention à la collision entre les 2 piles ! Ce sont les méthodes que la pile double devrait pouvoir exécuter :

Fonction	Signature	Détails
Push	public boolean push(boolean one, E e)	Ajoute un élément sur la pile et renvoie vrai. Renvoie faux si ce n'est pas possible.
Pop	public E pop(boolean one)	Retire le dernier élément sur la pile et le renvoie.
Top	public E top(boolean one)	Renvoie le dernier élément sur la pile.
Size	public int size(boolean one)	Renvoie la longueur de la pile.
Is Full	public boolean isFull()	Vérifie si la pile double est pleine.
Print	public void print()	Imprime-le contenue des 2 piles

Soumettre les fichiers « DoubleStack.java » and « ArrayDoubleStack.java » dans un dossier appelé « DoubleStack » dans Studium.

3.3 Pile spéciale (3 points)

Écrire une classe « SpecialArrayStack » qui fonctionne à l'aide d'un tableau générique (N'oubliez pas que vous devez également implémenter son interface appelée « SpecialStack »). Le nombre maximal d'éléments est de 100. La pile spéciale devrait implémenter une méthode getMax() qui devrait renvoyer l'élément maximum stocké dans la pile spéciale en O(1) temps et O(1) espace supplémentaire, en plus des mêmes méthodes qui ont été implémentées dans «Pile simple».

Soumettre les fichiers « SpecialStack.java » et « SpecialArrayStack.java » dans un dossier appelé « SpecialStack » dans Studium.

3.4 Problèmes avec Piles (6 points) :

Problème 1 (3 points):

Le royaume d'Um est situé sur une planète extrêmement froide. L'hémisphère nord est plus chaud la première partie de l'année et l'hémisphère sud la seconde moitié. Cette situation étrange oblige ses habitants à déplacer toute la ville de la province de Nam à la province de Sam dans un délai de 10 jours. Toutefois, pour effectuer cette migration, ils doivent respecter les règles suivantes :

- Les villes sont construites sous terre, donc les gens qui vivent plus près de la surface doivent déplacer leur partie de la ville en premier (section en surface) et le niveau au fond est le dernier à partir.
- L'ordre actuel des sections de la ville doit être respecté lorsque la ville est complètement reconstruite à Sam (c'est une règle placée par le roi d'UM).
- Ils ne peuvent pas déplacer tous les citoyens à la fois de la ville de Nam à la ville de Sam, en raison de la météo brutale, ils doivent parfois utiliser la ville temporelle de Pam placée entre Nam et Sam pour faire le déplacement et rétablir l'ordre original de la ville.
- Le déplacement de chaque section (aller d'une ville à l'autre) prend une journée et un seul mouvement par jour est permis.

Le royaume d'Um s'est développé au cours des dernières années et le roi lui-même a demandé d'analyser si la migration de cette année serait un succès. Il est inquiet, car il vit dans la partie la plus chaude de la ville, mais aussi la dernière à se déplacer. Il sait que vous êtes un bon programmeur et que vous avez récemment mis en place une sorte de structure de données qui ressemble à la façon dont les villes d'Um sont assemblées et démontées. Votre tâche est de dire au roi combien de jours sont nécessaires pour déplacer le royaume entier et rétablir l'ordre original de la ville et s'ils seront en mesure de faire le mouvement dans l'espace de 10 jours. L'organisation actuelle du Nam est la suivante :

Top

“N1: Agriculture”
“N2: Manufacture”
“N3: Académie”
“N4: Gouvernement”
“N5: Le Roi”

Bottom

Pour résoudre ce problème, vous devez utiliser l’implémentation « ArrayStack » et ses opérations. Le programme doit retourner la réponse à la question du roi en lui donnant le statut prévu du déménagement dans chaque ville à la fin de chaque journée. Ensuite, à la fin de la simulation il doit retourner un message comme ceci : «Il est <POSSIBLE/NON POSSIBLE> de déplacer la ville dans les 10 jours, car <X> nombre de jours sont nécessaires.». Example :

Sorties :

« Jour 0 : Nam (<Sections actuelles en Nam>), Pam (<Sections actuelles en Pam>) et Sam (<Sections actuelles en Sam>) »

« Jour 1: Nam (<Sections actuelles en Nam>), Pam (<Sections actuelles en Pam>) et Sam (<Sections actuelles en Sam>) »
…
« Jour n: Nam (<Sections actuelles en Nam>), Pam (<Sections actuelles en Pam>) et Sam (<Sections actuelles en Sam>) »

«Il est <POSSIBLE/NON POSSIBLE> de déplacer la ville dans les 10 jours, car <X> nombre de jours sont nécessaires.».

Enfin, vous devrez écrire dans votre rapport l’algorithme que vous avez utilisé pour déplacer la ville et relancer l’ordre. Quelle est sa complexité temporelle et spatiale ?

Soumettre les fichiers « MovingDay.java » et « Main.java » dans un dossier appelé «MovingDay » dans Studium.

Problème 2 (3 points) :

Créez une classe appelée « DuplicateEater » et à l'intérieur de celle-ci la méthode « pairDestroyer » qui prend en entrée un tableau générique avec une séquence de mots. Cette méthode vérifie si deux mots similaires se suivent et les élimine. À la fin de son processus de validation et de destruction, il imprimera le nombre de mots laissés dans la séquence après cette destruction par paires. Vous devez implémenter la fonction en utilisant l'implémentation « ArrayStack ». Incluez dans le rapport la complexité temporelle et spatiale de cette fonction.

Voici quelques exemples du comportement attendu :

Entrée : pile, liste, liste, file, arbre
Sortie : 3

Entrée : arbre, pile, pile, arbre
Sortie : 0

Soumettre les fichiers « DuplicateEater.java » dans un dossier appelé « DuplicateEater » dans Studium.